Transient fault-tolerance primitives

George Polevoy

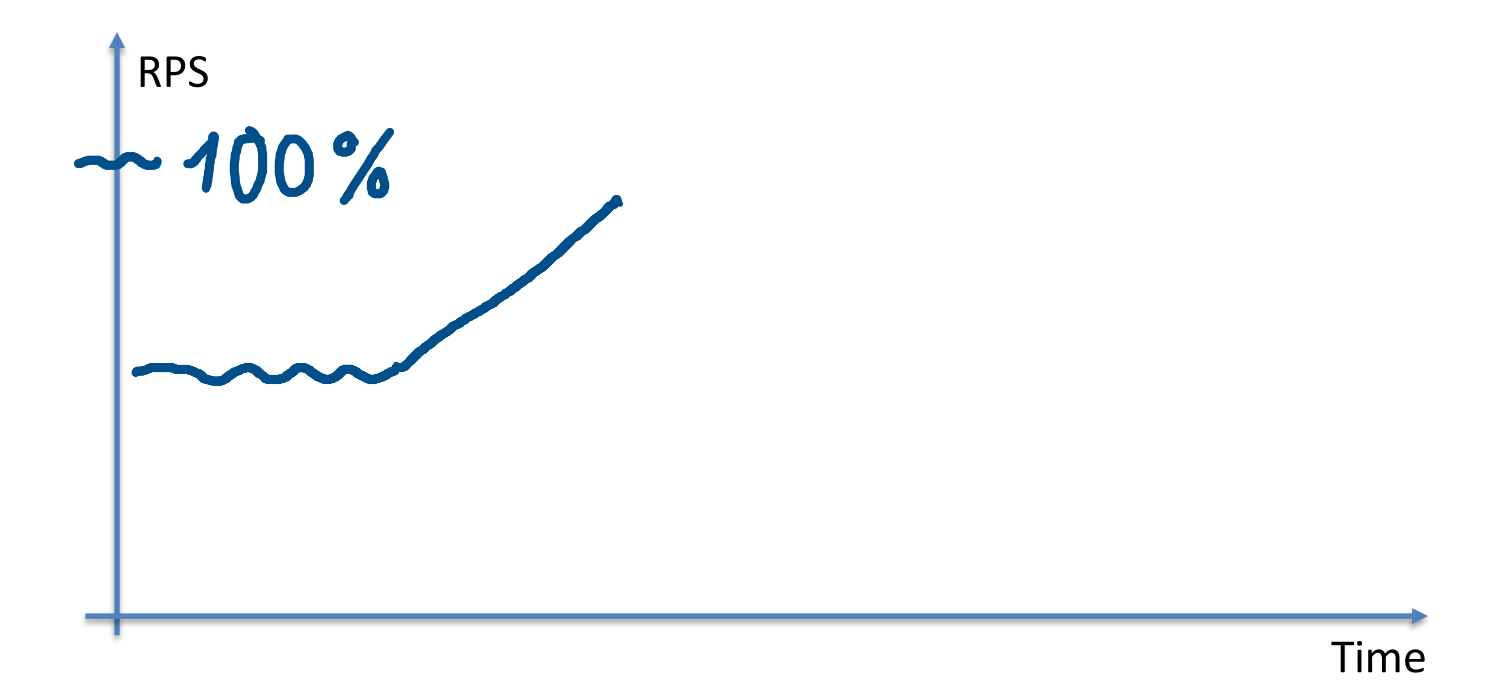
Dodo Engineering






RPS
Time


RPS
Time


RPS
~100%
Time

RPS

~100%

Time



RPS

~100%

~12%

Time



RPS

~100%

Time



A million dollar question

Should the server ”die” under the load?



1500
1000
500
0

10 20 30 40 50

time



300
200
100

300
200
100
0
10
15
20
25
30
time



600
400
200
0
30 35 40 45 50 55
time



Little Little's formula

$$L = \lambda W$$

- L - number of customers inside (*concurrency*)
- λ – arrival rate
- W – time inside



Requests, Served, Missed, Concurrency & Wasted

A(t) - arrivals

D(t) - departures

concurrency=

$$\frac{1}{T}\int\left(A(t)-D(t)\right)dt$$

1500
1000
500
0
10 20 30 40 50
time

Requests
Served
Missed
Concurrency
Wasted



A(t) - arrivals

D(t) - departures



concurrency=

$$1/T \int \left(A(t) - D(t) \right) dt$$

$$A(t) > D(t) \geq 0$$

$$A(t) \approx C$$




Concurrency

Concurrency is not parallelism

“Concurrency is about dealing with multiple things at the same time, parallelism is about doing multiple things at the same time”

Rob Pike

https://blog.golang.org/concurrency-is-not-parallelism



Accumulating concurrency

Game 1

Fair Scheduling

Workload: 3

Parallelism: 2



Accumulating concurrency

request number

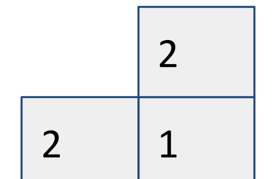
2

time



Accumulating concurrency

request number



time



Accumulating concurrency

request number

2.33

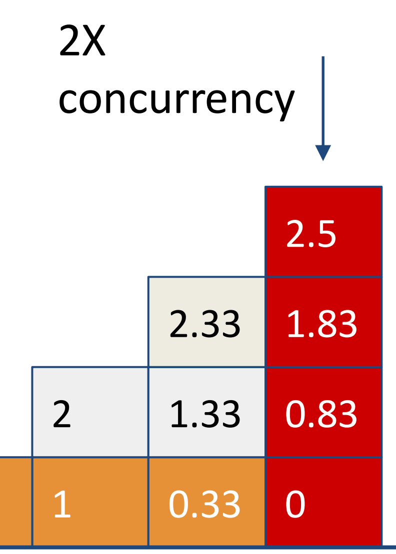
2 1.33

2 1 0.33

time



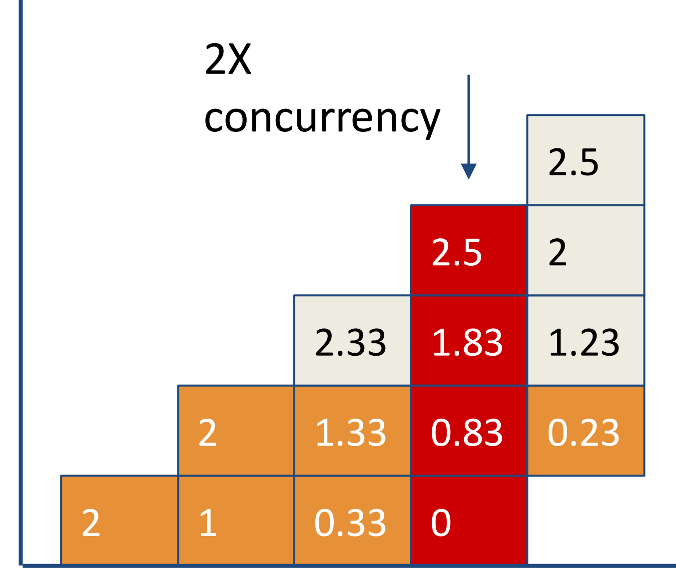
Accumulating concurrency





Accumulating concurrency

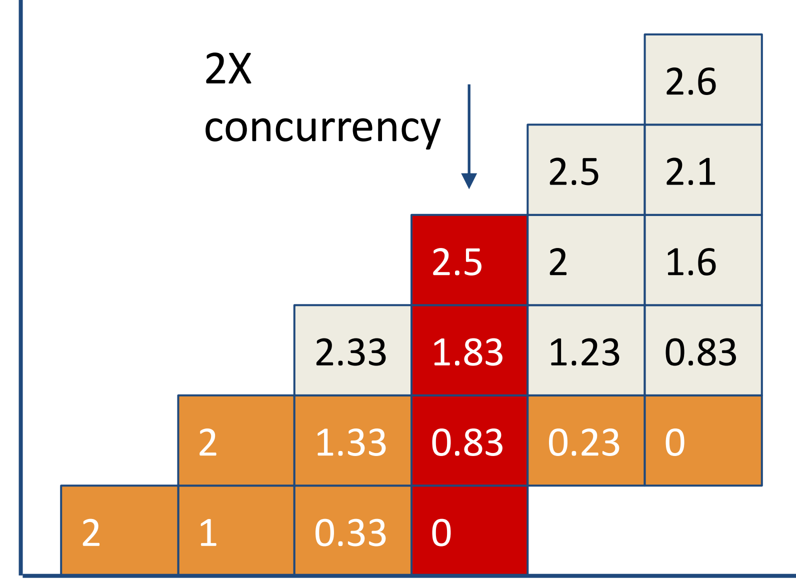
request number



time



Accumulating concurrency





request number

2X
concurrency

							2.66
						2.6	2.27
					2.6	2.2	1.87
				2.5	2.1	1.7	1.37
			2.5	2	1.6	1.2	0.87
		2.33	1.83	1.23	0.83	0.43	0.1
	2	1.33	0.83	0.23	0		
2	1	0.33	0				

2x response time

time



request number

2X
concurrency

								2.71
							2.66	2.37
						2.6	2.27	1.98
					2.6	2.2	1.87	1.58
				2.5	2.1	1.7	1.37	1.08
			2.5	2	1.6	1.2	0.87	0.58
		2.33	1.83	1.23	0.83	0.43	0.1	0
	2	1.33	0.83	0.23	0			
2	1	0.33	0					

2x + response time

time



Accumulating concurrency

Game over!






Nightclub bouncer pattern

Controlled concurrency

Game 2

Concurrency Limit = 2

Queue Size = 2



Controlled concurrency

request number

2

time



Controlled concurrency

request number

2

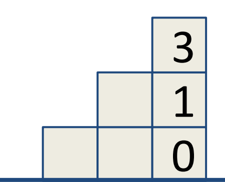
1

time



Controlled concurrency

request number

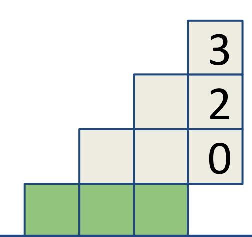


time



Controlled concurrency

request number



time



Controlled concurrency

request number

3
2
1

time



Controlled concurrency

request number

3
3
1
0

time

Controlled concurrency

request number

2X
concurrency

3
3
2
0

time



Controlled concurrency

request number

2X
concurrency

3
3
2
1

time



Controlled concurrency

request number

2X
concurrency

								3
								3
								1
								0

time



request number

2X
concurrency

3
3
2
0

time



request number

2X
concurrency

3
3
2
1

time



request number

2X
concurrency

3
3
1
0

time



request number

3
3
2
0

2X
concurrency

2X response
time

time



request number

3

3

2

1

2X concurrency

time



request number

2X
concurrency

3
3
1
0

time



Bulkhead

```
const int MaxConcurrency = 100;
SemaphoreSlim bulkhead = new SemaphoreSlim(MaxConcurrency, MaxConcurrency);
//…
  if (!await bulkhead.WaitAsync(TimeSpan.FromSeconds(1.0)))
  {
    throw new Exception(”Bulkhead rejected”);
  }
  try { await ProcessRequestInternal(); return; }
  finally { bulkhead.Release(); }
```



Bulkhead

```
var bulkhead = Policy.Bulkhead(100, 100);

//...

bulkhead.Execute(…);
```



Bulkhead (Polly library)

```
private const int inside = 100;
private const int outside = 100;

private static readonly BulkheadPolicy bulkhead =
    Policy.BulkheadAsync(inside, outside);

//...
try
{
    await bulkhead.ExecuteAsync(…);
}
catch (BulkheadRejectedException) { … }
```



1500
1000
500
0
10 20 30 40 50
time



100 inside, 100 outside

1000
500
0
10 20 30 40 50
time



100 inside, 500 outside

1000

500

0

10 20 30 40 50

time



100 inside, 1000 outside

1000
500
0
10
20
30
40
50
time



100 inside, 100 outside

500
0
10
20
30
40
50
time



Thread Pool regulates concurrency

GC
pause?

20
0

20
0

Thread
Pool
Threads
~20

Queue:
~200

..
..
..

Thread 1
Thread
...
Thread
20

Conn
pool
20



Async restores concurrency

200

0

async tasks,
Size:
∞

Conn pool
200



ONE DOES NOT SIMPLY

RETRY



Retry induced failure

O(N)

Total number of requests

Min response time

Recovery interval



Calculating total time of exponentail

$$\sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}.$$

total = first (2^n - 1)

first = total / (2^n - 1)



Lowering the number of requests

Total number of requests lowered

Min response time is the same

Recovery interval is ok



Concurrency is lower using Exponential Backoff

8000
6000
4000
2000
0
10
20
30
40
50
time

O(N^2)

O(N log(N))



Exponential Backoff

```
Policy
    .Handle<HttpRequestException>()
    .WaitAndRetry(5,
        retryAttempt =>
            TimeSpan.FromSeconds(Math.Pow(2, retryAttempt));
```



Fixing concurrency spikes

```
i => Math.Pow(2, i)
```

100000
80000
60000
40000
20000
0

```
i => random.NextDouble() + Math.Pow(2, i)
```

10000
8000
6000
4000
2000
0



Randomization is not trivial

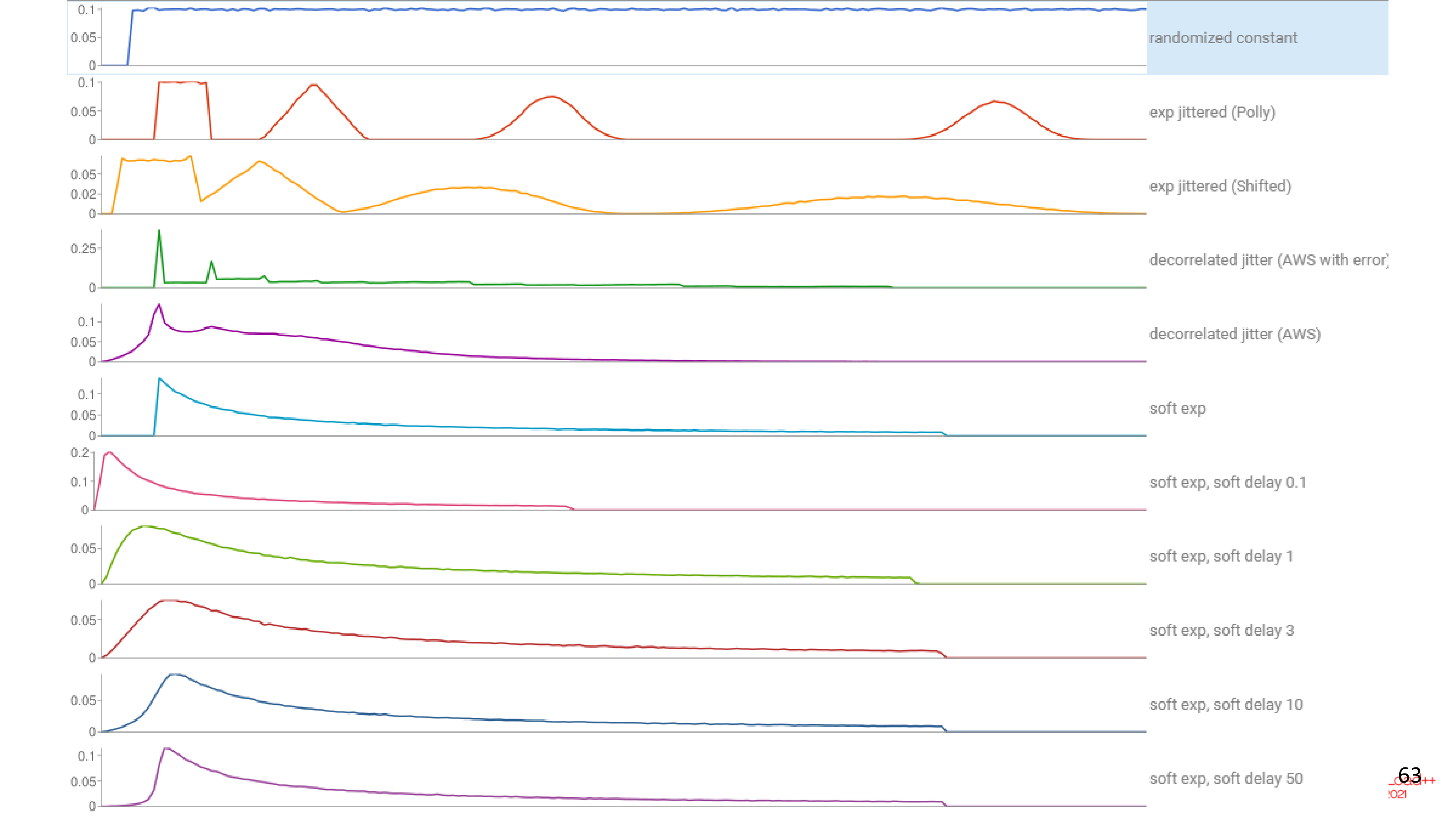
no jitter 100

naive 10

true decorrelated 7

wrong jitter 30




0.1
0.05
0
randomized constant
0.1
0.05
0
exp jittered (Polly)
0.05
0.02
0
exp jittered (Shifted)
0.25
0
decorrelated jitter (AWS with error)
0.1
0.05
0
decorrelated jitter (AWS)
0.1
0.05
0
soft exp
0.2
0.1
0
soft exp, soft delay 0.1
0.05
0
soft exp, soft delay 1
0.05
0
soft exp, soft delay 3
0.05
0
soft exp, soft delay 10
0.1
0.05
0
soft exp, soft delay 50

Decorrelated Jitter

```
IEnumerable<TimeSpan> DecorrelatedJitter()
{
   for (var softAttemptCount = 0.0; softAttemptCount < 4; )
   {
      softAttemptCount += r.NextDouble() * 2;
      yield return TimeSpan.FromSeconds( Math.Pow(2, softAttemptCount) * random.NextDouble());
   }
}
```



Decorrelated Jitter

```
var delay = Backoff.DecorrelatedJitterBackoffV2(
    medianFirstRetryDelay: TimeSpan.FromSeconds(1),
    retryCount: 5);

var retryPolicy = Policy .Handle<FooException>() .WaitAndRetryAsync(delay);
```



New Polly.Contrib.WaitAndRetry jitter formula - median initial delay 1 second, 5 tries

0.12
0.1
0.08
0.06
0.04
0.02
0

0 2 4 6 8 10 12



Complex policy using Polly

```
sc.AddHttpClient<IService, Client>(client => { client.Timeout = settings.TimeOutPerRequest; })
.AddPolicyHandler(
   Policy
   .TimeoutAsync<HttpResponseMessage>(settings.TotalTimeOut))
.AddPolicyHandler(
   HttpPolicyExtensions
   .HandleTransientHttpError()
   .Or<TimeoutRejectedException>()
   .WaitAndRetryAsync(settings.RetryCount,
      i => TimeSpan
      .FromMilliseconds(20 * Math.Pow(2, i))))
// …
```



Complex policy using Polly

```
// …
.AddPolicyHandler(
   Policy
   .TimeoutAsync<HttpResponseMessage>(settings.TimeOutPerRequest))
.AddPolicyHandler(
   HttpPolicyExtensions
   .HandleTransientHttpError()
   .AdvancedCircuitBreakerAsync(
      settings.FailureThreshold,
      settings.SamplingDuration,
      settings.MinimumThroughput,
      settings.DurationOfBreak));
```



What about a Cirquit Breaker?





Cold Start failure

20000
15000
10000
5000
0

10
15
20
25

Requests
Served
Missed
Concurrency
Wasted

1000



Beware - concurrency accumulates!

3000
2000
1000
0

0 20 40 60

time

Requests
Served
Missed
Concurrency
Wasted



A stop can rise concurrency!

2500
2000
1500
1000
500
0

10
15
20
25

1000

Requests
Served
Missed
Concurrency
Wasted



High contention code

Threads

Network

DB

Contended Key

Less important operation goes first

Lock

Thread is sleeping



Fallback for secondary operation

```
var fallback = FallbackPolicy<OptionalData>
  .Handle<OperationCancelledException>()
  .FallbackAsync<OptionalData>(OptionalData.Default);

var optionalDataTask = fallback
  .ExecuteAsync(async () => await CalculateOptionalDataAsync());

//…

var required = await CalculateRequiredData();

var optional = await optionalDataTask;

var price = CalculatePriceAsync(optional, required);
```



Threads

Network

DB

Contended Key

Threads

Network

DB

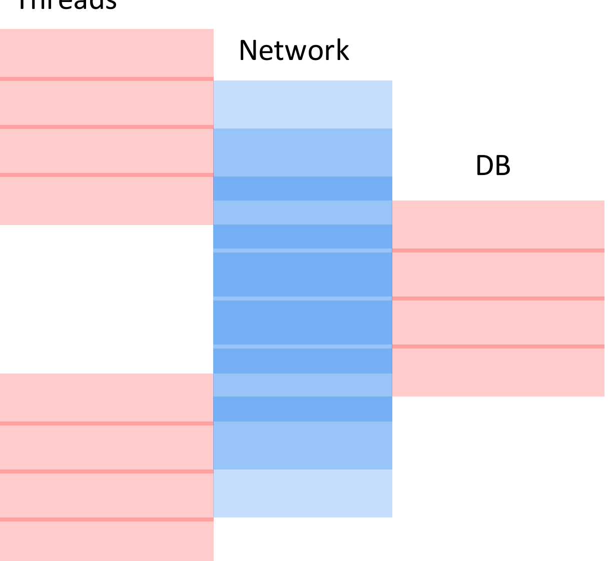


Threads

Network

DB

Contended Key





Threads

Network

DB

Contended Key

Threads

Network

DB



Summary

- Concurrency needs to be controlled

What affects concurrency?

- External load
- Retry
- Temporary stop
- Contention



Георгий Полевой
Dodo Engineering
george.polevoy@gmail.com

https://t.me/dododev
https://dodo.dev



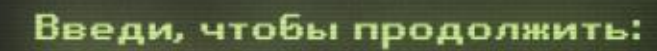
DODO ENGINEERING © 2021

Hi %username%

Dodo Engineering still hasn't you.
Follow the RabbitMQ.

Мы – разработчики в Dodo: «blameless culture» + «no bullshit». Боремся с англицизмами, используем их и снова боремся. Создаём систему Dodo IS. Она — симбиоз ERP, HRM и CRM-систем. Она управляет всем бизнесом Dodo.

Мы работаем в 13 странах. В планах весь мир. Нам нужен ты.



> business
> stack
> jobs
> projects
> test
> help

dodo>

Пользовательское соглашение





SOUND




HL
HighLoad++
Весна 2021